What is Model Pruning?

- Technique of removing connections between neurons.
- Sometimes weights are removed and sometimes entire channels or filters are removed.

Why Model Pruning?

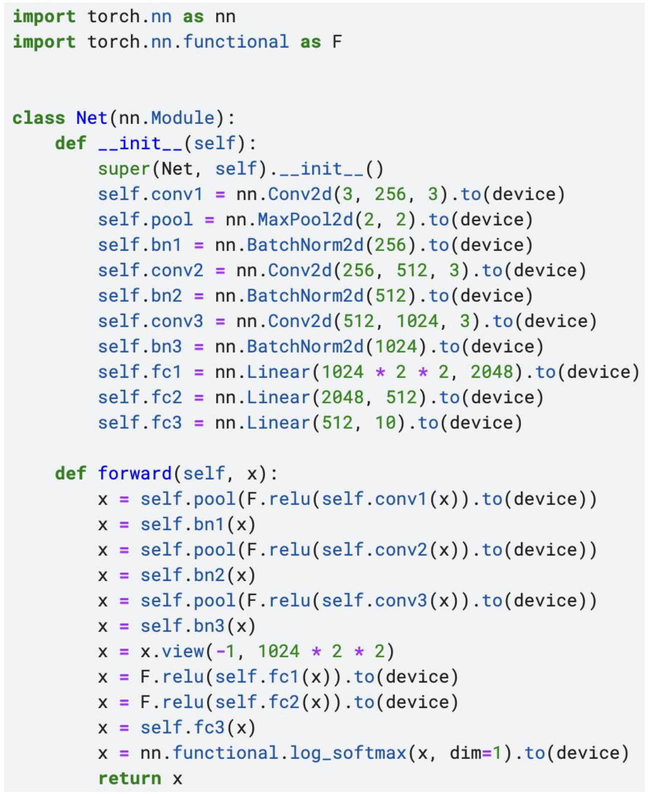
- Deep neural networks tend to be over-parameterized.
- Multiple features convey the same information and thus we can get rid of these redundancies.
- Reduces inference time and inference cost.

Types of pruning:

- **Unstructured Pruning:**
  - Involves removal of individual weights
  - Results in much higher sparsity
  - Requires specialised hardware/software to see performance gains.
- **Structured Pruning:**
  - Involves the removal of entire filters/channels.
  - Lesser sparsity than unstructured pruning
  - More challenging but offers performance gains “off the shelf”.

Some basic experiments using built-in PyTorch pruning utilities:

- Architecture of model used:



```
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 256, 3).to(device)
        self.pool = nn.MaxPool2d(2, 2).to(device)
        self.bn1 = nn.BatchNorm2d(256).to(device)
        self.conv2 = nn.Conv2d(256, 512, 3).to(device)
        self.bn2 = nn.BatchNorm2d(512).to(device)
        self.conv3 = nn.Conv2d(512, 1024, 3).to(device)
        self.bn3 = nn.BatchNorm2d(1024).to(device)
        self.fc1 = nn.Linear(1024 * 2 * 2, 2048).to(device)
        self.fc2 = nn.Linear(2048, 512).to(device)
        self.fc3 = nn.Linear(512, 10).to(device)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)).to(device))
        x = self.bn1(x)
        x = self.pool(F.relu(self.conv2(x)).to(device))
        x = self.bn2(x)
        x = self.pool(F.relu(self.conv3(x)).to(device))
        x = self.bn3(x)
        x = x.view(-1, 1024 * 2 * 2)
        x = F.relu(self.fc1(x)).to(device)
        x = F.relu(self.fc2(x)).to(device)
        x = self.fc3(x)
        x = nn.functional.log_softmax(x, dim=1).to(device)
        return x
```

Total number of trainable parameters: 15355402

- Purpose: Classification on the cifar10 dataset
- Number of classes: 10
- Libraries used:
  - torch
  - torchvision
  - torch-pruning
- Accuracy of the unpruned pre-trained model: 79.67%
- Total inference time on test data, for the unpruned pre-trained model: 1310.764 ms
- Total model size of the unpruned pre-trained model: 61.45 MB

Results:

	Pruning type	Description	Units pruned	Function calls	No. of parameters pruned	% of parameters pruned	Accuracy after pruning	Accuracy drop after pruning

1	**Random Unstructured**	Randomly prune 40% of the connections	Conv1's weight & bias
			
			Conv1's weight
				module = net2.conv1
				prune.random_unstructured(module, name="weight", amount=0.4)
				prune.random_unstructured(module, name="bias", amount=0.4)
					2867
					
					2765
						0.0187
						
						0.018
							21.66%
							
							34.88%
								58.01%
								
								44.79%

2	**L1 Unstructured**	Prune 40% of the connections based on minimum L1 norm	Conv1's weight & bias
			
			Conv1's weight
				module = net3.conv1
				prune.l1_unstructured(module, name="weight", amount=0.4)
				prune.l1_unstructured(module, name="bias", amount=0.4)
					2867
					
					2765
						0.0187
						
						0.018
							71.45%
							
							72.13%
								8.22%
								
								7.54%

3	**Random Structured Pruning**	Randomly prune x% of the channels/filters	Conv1's weight	module = net4.conv1 prune.random_structured(module, name="weight", amount=0.41479, dim=0)	2754 **(40% pruned)** 2862 **(41.479% pruned)**	0.0179 0.0186	66.78% 70.27%	12.89% 9.4%
4	**Ln Structured Pruning**	Pruning x% of the channels based on the channels' Ln norm	Conv1's weight	module = net5.conv1 prune.ln_structured(module, name="weight", amount=0.41479, n=1, dim=0)	2754 **(n=2, 40% pruned)** 2862 **(n=1, 41.479% pruned)**	0.0179 0.0186	75.11% 74.04%	4.56% 5.63%

5	**Pruning multiple parameters**	Using l1_unstructured, prune 30% of connections in all Conv2D layers and prune 40% of the connections in all Linear layers	All Conv and Linear layers' weight and bias	for name, module in net6.named_modules():
				    if isinstance(module, torch.nn.Conv2d):
				        prune.l1_unstructured(module, name='weight', amount=0.3)
				        prune.l1_unstructured(module, name='bias', amount=0.3)
				    elif isinstance(module, torch.nn.Linear):
				        prune.l1_unstructured(module, name='weight', amount=0.4)
				        prune.l1_unstructured(module, name='bias', amount=0.4)
					5550033	36.144	76.99%	2.68%

6	**Global Pruning**	Pruning 75% of connections across the whole model using l1_unstructured	All layers' weight and bias	parameters_to_prune = ((net7.conv1, 'weight'), (net7.bn1, 'weight'), ……, (net7.fc3, 'bias')) prune.global_unstructured(parameters_to_prune, pruning_method=prune.L1Unstructured, amount=0.75,)	11516552	75.00	69.16%	10.51%

7	**Custom Pruning**	A unstructured pruning technique that prunes every other entry in a tensor	Conv1's weight and bias	def custom_unstructured(module, name):
				    CustomPruningMethod.apply(module, name)
				    return module
				custom_unstructured(net8.conv1, name='weight')
				custom_unstructured(net8.conv1, name='bias')
					3584	0.0233	17.13%	62.54%

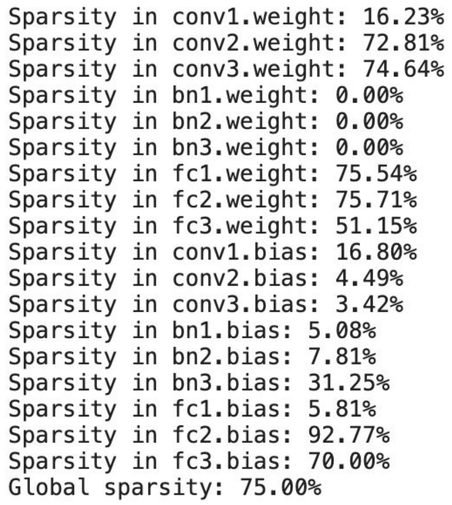
Note: After modifying the architectures of the models pruned using structured pruning, the following results were obtained:

- Random Structured Pruning → 70.27% test accuracy, 1145.376 ms total test inference time, 59.55 MB model size
- L1 Structured Pruning → 72.36% test accuracy, 1175.0053 ms total test inference time, 59.55 MB model size

Analysis and Observations:

- The ratio of important to unimportant weights is much higher in lower Conv layers as compared to deeper Conv layers.

- A comparison between accuracy drops of structured and unstructured pruning:
  - Random Unstructured pruning results in a much higher accuracy drop as compared to Random Structured pruning.
  - This is unlike the pruning performed based on L1 norm, where there is negligible difference between accuracy drops of structured and unstructured variants.



```
Sparsity in conv1.weight: 16.23%
Sparsity in conv2.weight: 72.81%
Sparsity in conv3.weight: 74.64%
Sparsity in bn1.weight: 0.00%
Sparsity in bn2.weight: 0.00%
Sparsity in bn3.weight: 0.00%
Sparsity in fc1.weight: 75.54%
Sparsity in fc2.weight: 75.71%
Sparsity in fc3.weight: 51.15%
Sparsity in conv1.bias: 16.80%
Sparsity in conv2.bias: 4.49%
Sparsity in conv3.bias: 3.42%
Sparsity in bn1.bias: 5.08%
Sparsity in bn2.bias: 7.81%
Sparsity in bn3.bias: 31.25%
Sparsity in fc1.bias: 5.81%
Sparsity in fc2.bias: 92.77%
Sparsity in fc3.bias: 70.00%
Global sparsity: 75.00%
```

Global Pruning sparsity report

Note: Ideally, in structured pruning, we are supposed to prune not only the channels of the 'weight' parameter, but also the corresponding channels of the 'bias' and the appropriate entries of all the filters of the next layer. For most of the experiments performed above, only the 'weight' parameter has been pruned (i.e. zeroed out).

For more information please refer:

- https://pytorch.org/tutorials/intermediate/pruning_tutorial.html
- https://github.com/Riddhi-Chatterjee/Model-Pruning/tree/main/Basic-Experiments

Some existing model pruning approaches:

- **Based on minimum weights:** Prune the weights/filters with lower norm. A unit with a lower norm detects less important features than that having a higher norm. This approach can be aided by applying L1 or L2 regularisation during training, which will force unimportant kernels to have smaller values.

- **Based on activation:** If the mean activation value corresponding to some filter is small then this filter isn’t an important feature detector for the task at hand.

- **Based on Average Percentage of Zeros (APoZ) value:** Activation functions like ReLU imposes sparsity during inference. Thus average percentage of zero activation values at the output can be used to determine the importance of the channel/filter.

APoZ can only be used for structured pruning. The rest of the above approaches can be used for both unstructured and structured pruning.

Literature review on Model Pruning:

1. TVSPrune — Pruning Non-Discriminative Filters Via Total Variation Separability Of Intermediate Representations Without Fine Tuning

- Structured pruning
- No access to training dataset or loss function – Distributional Pruning
- For generalization purposes it is sufficient to retain those filters which discriminate well between classes.
- **Discriminative Filters:** Class conditional distributions of its outputs are “well separated”
  Criterion used: Least TV Distance between pairs of classes.

2. Pruning Convolutional Neural Networks for Resource Efficient Inference

Link to code repo: https://github.com/insomnia250/channel-prune

- Can be used for both Structured and Unstructured pruning.
- For a set of training examples D, the network's parameters W are optimised to minimise a cost value C(D|W). The objective is as follows:

$$\min_{\mathcal{W}'} \left| \mathcal{C}(\mathcal{D}|\mathcal{W}') - \mathcal{C}(\mathcal{D}|\mathcal{W}) \right| \quad \text{s.t.} \quad ||\mathcal{W}'||_0 \leq B$$

- Iteratively identify and remove the least important parameters — ensures the eventual satisfaction of the bound on the number of non-zero parameters in W'.
- A feature map corresponding to the lth layer and kth channel is computed as follows:

$$\mathbf{z}_\ell^{(k)} = \mathbf{g}_\ell^{(k)} \mathcal{R}\left(\mathbf{z}_{\ell-1} * \mathbf{w}_\ell^{(k)} + b_\ell^{(k)}\right)$$

where, g, w, and b denote the pruning gate, weights and bias of the lth layer and kth channel respectively.

- Using Taylor expansion, we approximate the change in the loss value resulting from the removal of a particular parameter. For a feature map, this turns out to be as follows:

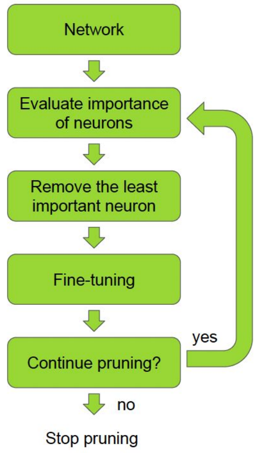
$$\Theta_{TE}(z_l^{(k)}) = \left| \frac{1}{M} \sum_m \frac{\delta C}{\delta z_{l,m}^{(k)}} z_{l,m}^{(k)} \right|$$

- A simple layer-wise L2 normalisation can achieve adequate rescaling across layers:

$$\hat{\Theta}(\mathbf{z}_l^{(k)}) = \frac{\Theta(\mathbf{z}_l^{(k)})}{\sqrt{\sum_j \left(\Theta(\mathbf{z}_l^{(j)})\right)^2}}$$

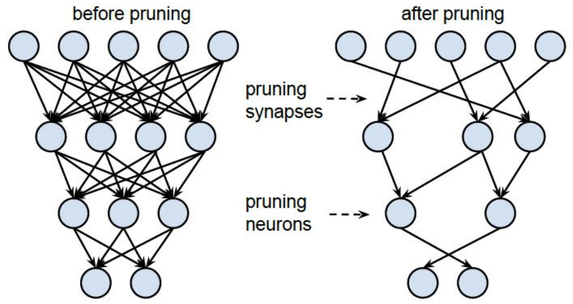
- FLOPs regularization:

$$\Theta(\mathbf{z}_l^{(k)}) = \Theta(\mathbf{z}_l^{(k)}) - \lambda \Theta_l^{flops}$$

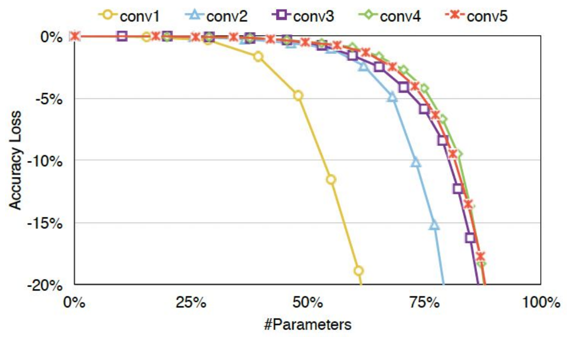

Network
Evaluate importance of neurons
Remove the least important neuron
Fine-tuning
Continue pruning?
yes
no
Stop pruning

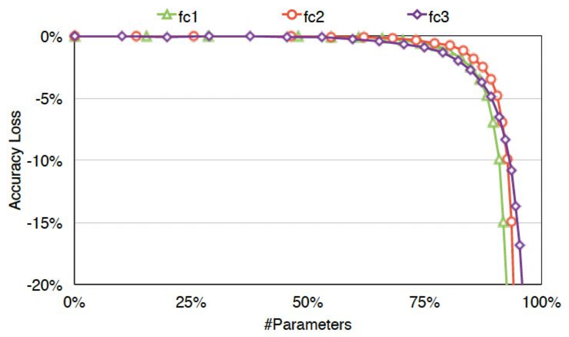
3. Learning both Weights and Connections for Efficient Neural Networks

- Can be used for both Structured and Unstructured pruning.
- After an initial training phase, we remove all connections whose weight is lower than a threshold.
- We then retrain the sparse network so the remaining connections can compensate for the connections that have been removed.
- The phases of pruning and retraining may be repeated iteratively to further reduce network complexity.



- Using L1 regularisation results in better accuracy after pruning, but before re-training. The remaining connections are not as good as with L2 regularisation, resulting in lower accuracy after re-training. Overall, L2 regularisation gives the best pruning results.
- As pruning already reduces model capacity, the retraining dropout ratio should be smaller.
- When we retrain the pruned layers, we should keep the surviving parameters instead of re-initializing them.
- **Sensitivity Pruning:** We used the sensitivity results to find each layer's threshold:
  - For example, the smallest threshold was applied to the most sensitive layer, which is the first convolutional layer.

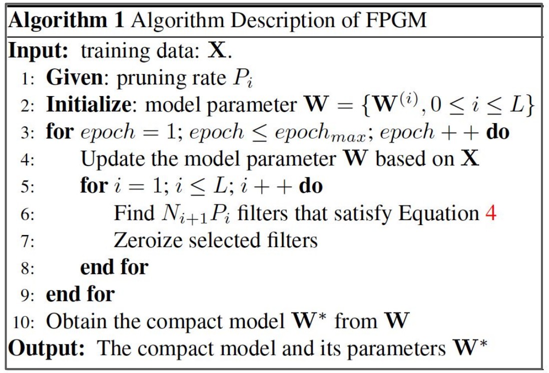




Pruning sensitivity for CONV layer (left) and FC layer (right) of AlexNet.

4. Filter Pruning via Geometric Median for Deep Convolutional Neural Networks Acceleration

Link to code repo: https://github.com/he-y/filter-pruning-geometric-median



Algorithm 1 Algorithm Description of FPGM

```
Input: training data: $\mathbf{X}$.
 1: Given: pruning rate $P_i$
 2: Initialize: model parameter $\mathbf{W} = \{\mathbf{W}^{(i)}, 0 \leq i \leq L\}$
 3: for $epoch = 1; epoch \leq epoch_{max}; epoch++$ do
 4:     Update the model parameter $\mathbf{W}$ based on $\mathbf{X}$
 5:     for $i = 1; i \leq L; i++$ do
 6:         Find $N_{i+1}P_i$ filters that satisfy Equation 4
 7:         Zeroize selected filters
 8:     end for
 9: end for
10: Obtain the compact model $\mathbf{W}^*$ from $\mathbf{W}$
Output: The compact model and its parameters $\mathbf{W}^*$
```

- Can be used for Structured pruning.
- The effectiveness of a "smaller-norm-less-important" based criterion for filter pruning depends on two requirements that are not always met: i) The norm deviation of the filters should be large. ii) The minimum norm of the filters should be small.
- FPGM gets rid of the above requirements — compresses CNN models by pruning "filters with redundancy".
- We calculate the Geometric Median (GM) of the filters within the same layer. According to the characteristics of GM, the filter(s) F near it can be represented by the remaining ones. Therefore, pruning those filters will not have substantial negative influences on model performance.
- For a set of points in d-dimensional space, the geometric median is defined to be a point in d-dimensional space that minimises the sum of Euclidean distances to them. The geometric median represents the common information among all the filters of the ith layer:

$$x^{GM} = \underset{x \in \mathbb{R}^{N_i \times K \times K}}{\arg\min} \sum_{j' \in [1, N_{i+1}]} \|x - \mathcal{F}_{i,j'}\|_2.$$

- In the ith layer, the filter(s) nearest to the geometric median are found as follows:

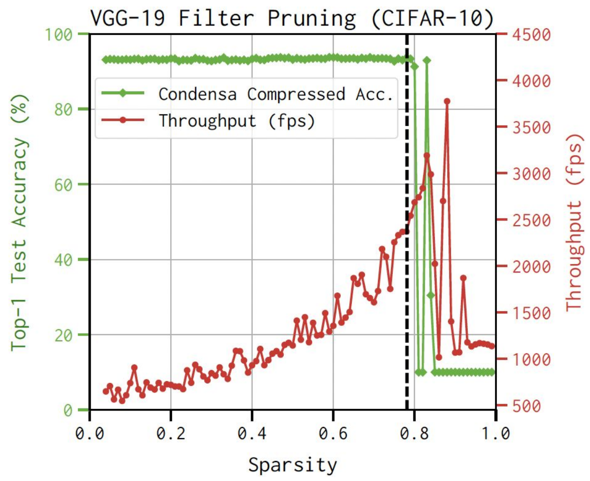
$$\mathcal{F}_{i,j^*} = \underset{\mathcal{F}_{i,j'}}{\arg\min} \|\mathcal{F}_{i,j'} - x^{GM}\|_2, \text{ s.t. } j' \in [1, N_{i+1}],$$

- The above set of filter(s) can be represented by other filters of that layer, and thus pruning them results in negligible accuracy drop.
- Finding the geometric median is computationally expensive. We could instead find the filter which minimises the summation of the distances from other filters.

5. A Programmable Approach to Neural Network Compression

Link to code repo: https://github.com/NVlabs/condensa

- Supports both Structured and Unstructured pruning.
- Users programmatically compose simple operators, in Python, to build more complex and practically interesting compression strategies.
- Given a strategy and user-provided objective (such as minimization of running time), Condensa uses a novel Bayesian optimization-based algorithm to automatically infer desirable sparsities.
- The prune operator — for unstructured magnitude-based pruning.
- The filter_prune, neuron_prune, and blockprune operators — for structured pruning.
- Each operator can be applied on a per-layer basis.
- Condensa requires:
    - A working Linux installation (Ubuntu 18.04 has been used)
    - NVIDIA drivers and CUDA 10+ for GPU support
    - Python 3.5 or newer
    - PyTorch 1.0 or newer



```python
# Construct pre-trained model
criterion = torch.nn.CrossEntropyLoss()
train(model, num_epochs, trainloader, criterion)

# Instantiate compression scheme
prune = condensa.schemes.FilterPrune()
# Define objective function
tput = condensa.objectives.throughput
# Specify optimization operator
obj = condensa.searchops.Maximize(tput)
# Instantiate L-C optimizer
lc = condensa.optimizers.LC(steps=30, lr=0.01)
# Build model compressor instance
compressor = condensa.Compressor(
    model=model, # Trained model
    objective=obj, # Objective
    eps=0.02, # Accuracy threshold
    optimizer=lc, # Accuracy recovery
    scheme=prune, # Compression scheme
    trainloader=trainloader, # Train dataloader
    testloader=testloader, # Test dataloader
    valloader=valloader, # Val dataloader
    criterion=criterion # Loss criterion
)
# Obtain compressed model
wc = compressor.run()
```

Listing 1. Example usage of the CONDENSA library.